CSE 501 Project Report

Interprocedural Analysis in ZPL

A.J. Bernheim Doug Zongker

March 14, 1997

We have implemented an interprocedural analysis framework for the ZPL compiler. ZPL is a high-level, machine-independent, implicitly-parallel, array-based programming language geared toward scientific applications. ZPL is intended to replace languages such as Fortran and C for scientific computation [Sny96].

The current compiler has some interprocedural optimization, but no unified, accessible framework for adding new optimizations. We used our framework to implement alias analysis, parameter in-out analysis and the accompanying transformations. Several ZPL programs have been used to evaluate the success of our analyses and transformation. Since ZPL is a machine-independent language we have timed the ZPL programs on five different machines: the DEC Alpha, the Intel Paragon, the IBM SP/2 and the Cray T3D and T3E. For ZPL programs expected to benefit from our transformation we have achieved respectable speedups on all the machines.

The framework has also been designed with the goal of making implementing client analyses as easy as possible. This will encourage others to extend the compiler to include additional interprocedural optimizations.

Section 1 describes our framework, while section 2 outlines alias and parameter in/out analysis. Section 3 presents our transformation is and Section 4 evaluates our framework qualitatively and summarize the results of our transformation. Finally, Section 5 offers our conclusions.

1 Framework

Our framework facilitates top-down interprocedural analyses. It provides an abstraction for the process of traversing over the call graph, allowing the client code to annotate the AST at each function node. The framework detects cycles in the call graph, corresponding to recursion in the program, and uses the client code to iteratively find a fixpoint. The client is responsible for any

- Main framework procedures called by client code

```
void ip_initialize ( module_t *mod )
void * ip_analyze ( function_t *f, void *incontext, analysis_info *a )
```

  `analysis_info` is a structure containing function pointers to the callbacks comprising the client: the main intraprocedural analysis function, and utility functions for merging, and comparing contexts.

- Annotating the call graph

```
void add_annotation ( function_t *f, char *id, void *data )
void *lookup_annotation ( function_t *f, char *id )
void delete_annotation ( function_t *f, char *id, int freedata )
```

- Traversing `f`'s call graph, calling back to client for each callsite

```
void traverse_callsites ( function_t *f, void (*c)(expr_t *, statement_t *) )
```

  (`module_t`, `function_t`, `expr_t` and `statement_t` are all structures defined by the ZPL compiler.)

Figure 1: Framework procedures called by the client.

intraprocedural analysis necessary, making callbacks to our framework for each call site. This structure is loosely based on the Vortex interprocedural analysis framework of [CDG96].

Two client defined structures are used for interprocedural analysis with our framework. The context structure built at each callsite contains the information to be passed to the analysis of the called function. To store the results of an analysis a client defines an annotation structure. The framework maintains a list of client annotation structures for each procedure. This allows new client analyses to be added without having to change the format of the AST's procedure nodes, which would necessitate rebuilding the entire ZPL compiler. Each annotation is named by a client-specified identifier string. A single client can add multiple annotations if desired. The framework provides functions to lookup, add, and delete annotations as necessary. The framework itself also uses this annotation mechanism to store internal information related to detecting recursive calls.

The client begins analysis by calling the framework's `ip_initialize()` function to clear any internal annotations already present for the module. The client then calls the `ip_analyze()` function, passing in a pointer to the entry procedure of the ZPL program, an initial context, and an `analysis_info` structure containing pointers to the client functions to be used for intraprocedural analysis and context manipulation. `ip_analyze()` is the main function of the framework; it is responsible for detecting and handling recursion. `ip_analyze()` first determines if the ZPL procedure is a library or external function, since these types of procedures can not be analyzed by

the client. Next, it checks to see if the ZPL procedure is currently in the midst of analysis (i.e., the current callback is a recursive call). If not, `ip_analyze()` marks the procedure as being under evaluation, stores a copy of the evaluation context, and passes control to the client callback.

If the framework detects recursion (the ZPL procedure is already undergoing analysis), then the framework must determine whether a fixpoint has been reached. It compares (using a client-provided `as_general_as()` function) the context that came in with the current call with the cached copy of the context that the procedure is already being evaluated with. If the cached context is as general as the new context, a fixpoint has been reached and the framework does not need to analyze the procedure any further. If not, then the new context contains new information. The two contexts are then merged (using a client-provided `merge()`) function, and the client callback is called with the merged context as its input.

For this scheme to work, the client callback must have a monotonic action on contexts. We view the space of possible contexts as a lattice, with the client's `as_general_as()` function forming the $\sqsubseteq$ relation. The client must ensure that when a procedure is analyzed recursively, the incoming context for the inner analysis is as general as (lower in the lattice than) the context of the outer analysis. There must also exist a bottom element of the lattice which is as general as any other element. If these conditions hold, then the analysis for recursive calls will always reach a fixpoint and terminate.

The framework also provides the `traverse_callsites()` function to simplify a client's traversal of a ZPL procedure looking for callsites. `traverse_callsites()` takes a ZPL procedure (a pointer to the corresponding procedure node in the AST) and a C function to be called at each callsite in that procedure. The C function is passed the current expression and containing statement as parameters. It calls the framework for the called procedure after constructing any necessary context for that callsite. `traverse_callsites()` allows the client to ignore the details of locating callsites within the AST for a ZPL procedure.

2 Analysis

We have implemented one client using this framework: alias analysis. Using the results of alias analysis, we then perform parameter in/out analysis. These results are used to determine when a parameter can be safely passed by reference. Currently the ZPL development group believes that in the presence of heavy aliasing the compiler may produce incorrect code due to an overly optimistic temporary insertion procedure. Accurate alias analysis would be useful not only for our in/out analysis, but also in improve correctness in other areas of the compiler.

2.1 Alias Analysis

Our alias analysis annotates each procedure with sets of variables (both globals and locals of ancestors in the call graph, one per formal) that can be passed in for each parameter. To implement

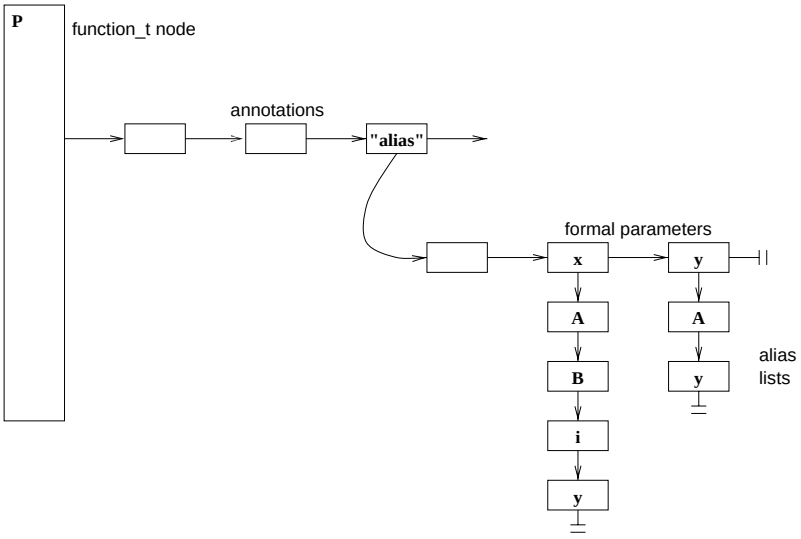

Figure 2: Structure of alias annotation for sample function P with formals x and y.

alias analysis as a client within the framework we first defined data structures for our annotations and contexts. For this analysis, the annotations (information stored by the client for each function) and contexts (information constructed to pass to a procedure call) use the same data structure: a set of variables that could be passed in for each formal. We provide the required functions for merging contexts, comparing context generality, and performing intraprocedural analysis. For alias analysis the `merge()` function performs per-parameter set union and `as_general_as()` function computes per-parameter superset: $A \sqsubseteq B$ if each set in A is a superset of the corresponding set in B. We took advantage of the framework's `traverse_callsites()` procedure to find callsites within ZPL procedures.

Our `alias_callback()` function which handles intraprocedural analysis first determines if an annotation from a previous evaluation of the procedure already exists. If no annotation exists, an annotation with an empty alias set for each formal is built. The annotation (whether it was just created or comes from a previous analysis of this procedure) is updated by merging it with the incoming context which contains, for each parameter, the set of variables that could be passed in at the current callsite. Figure 2 shows a sample annotation structure for alias analysis.

After the annotation for the current ZPL procedure has been computed, the function body is traversed using `traverse_callsites()`. Each callsite in the ZPL procedure is processed by the `handle_callsite()` function. This function looks up each actual which is a single variable (excluding expressions and constants) in the calling procedure's context and constructs the set of variables the actual could be aliased to. These sets form the calling context that is used to analyze the called ZPL procedure.

Our current implementation of alias analysis does some reevaluation of procedures that is

- Main function that issues first call to framework

```
static void do_module ( module_t *mod );
```

- Callback from framework for each function

```
static void * alias_callback ( function_t *f, void *incon );
```

- Callback from `traverse_callsite()` to construct calling contexts

```
static void handle_callsite ( expr_t *, statement_t * );
```

- Utility functions required by framework to manipulate contexts.

```
static void * alias_merge ( void *, void * );
static int alias_as_general_as ( void *, void * );
```

Figure 3: Client procedures for alias analysis.

strictly not necessary. This could be avoided by detecting when the current context used to evaluate a procedure is subsumed by the set of aliases already known for the procedure's formals.

2.2 Parameter In/Out Analysis

Our parameter in/out analysis determines when the contents of a parameter can be changed by a ZPL procedure. Passing large arrays is common in ZPL, so using in/out analysis to determine when a pass-by-value parameter can safely be changed to pass-by-reference could lead to large savings by eliminating the time needed for copying and the memory space for the temporary. Since formal procedure parameters in the language default to call-by-value, we expect unnecessary value parameters to appear frequently.

Parameter in/out analysis utilizes the information gathered in alias analysis to determine if a parameter can be passed by reference. A parameter can not be passed by reference if:

1. The formal parameter is assigned to in the callee procedure, or
2. The formal parameter is passed by reference from the callee procedure, or
3. A global in the set of variables that the callee's formal parameter could be aliased to is assigned to in the callee.

If none of these three types of possible assignments occur for a formal parameter, the parameter can be passed by reference without affecting the program's semantics.

With alias analysis already computed, parameter in/out analysis becomes an intraprocedural analysis that only needs to access the annotations left by alias analysis. Therefore the parameter

in/out analysis visits each procedure in the program once, and does not need to use the framework to traverse the call graph. However, parameter in/out analysis still uses the annotation utility functions provided by the framework to retrieve the alias annotations for each procedure. The client code for parameter in/out analysis consists of three C functions, `transform_varables()`, `find_assigned_s()` and `find_assigned_e()`.

`transform_varables()` is the main function for parameter in/out analysis. For each ZPL procedure, it first determines the set of variables that are assigned to within the procedure using `find_assigned_s()` and `find_assigned_e()`. These two functions traverse a procedure's statements and expressions, looking for places where a variable's value could be changed. Variables could change when they appear on the left side of an assignment expression, when they are passed by reference to other procedures, or when they appear in a `wrap`, `reflect`, or `flood` statements (these are ZPL-specific constructs for dealing with arrays).

Once the set of variables possibly modified within the ZPL procedure being analyzed has been determined, `transform_varables()` checks each formal parameter passed by value to determine if the parameter can be passed by reference using the three rules stated previously.

3 Transformation

Our transformation is conceptually simple. When parameter in/out analysis determines that a formal parameter can be passed by reference, we change the subclass of the parameter from `SC_VALUE` to `SC_REFER`, essentially adding the `var` keyword to the formal parameter declaration. Since our optimization pass runs before the compiler's temporary insertion pass, this change prevents the compiler from inserting a temporary.

There is a complication introduced by the fact that it is not legal in ZPL to pass an expression by reference. Whenever we change a parameter to pass-by-reference, we must insert temporaries at every callsite where the actual for that parameter is not a single variable—the temporary can then be passed by reference. This essentially undoes our transformation at that particular callsite. To save unnecessary work, we have an additional criterion for changing a parameter to pass-by-reference: there must be at least one callsite where a temporary is not needed.

4 Results

We evaluate the success of our analyses and transformation by measuring the impact of the optimization on compilation and execution times. Also, since the goal of implementing the analysis with a framework/client model was to facilitate the development of additional interprocedural analyses, we include some evaluations of the framework interface made by members of the ZPL team.

Benchmark	Compilation Time		
	Unoptimized (s)	Optimized (s)	Slowdown
embar	0.200	0.217	8.5%
fibro	9.617	22.333	132.2%
frac	0.183	0.150	-19.1%
pde1	0.250	0.267	6.8%
sp	23.450	24.333	3.7%
spuv	23.383	23.733	1.4%
test3	0.067	0.067	0.0%
zray	0.567	0.717	26.4%

Table 1: Compilation times of ZPL programs (in CPU seconds) with and without our optimization: alias analysis, parameter in/out analysis, and transformation.

4.1 Quantitative Evaluation

We are currently using several ZPL programs to evaluate the effects of our optimization. Our optimization only has the potential to improve performance when parameters are passed to procedures, so we have focused on five programs that we expect to benefit from our transformation. However, to validate the correctness of our analyses and transformation we have also run the compiler on several other benchmark programs that should not be altered by the transformation pass. For these types of programs we do not expect to see any significant speedup or slowdown, indicating that our optimization is not negatively impacting programs that can not be improved by our transformation. Two of these programs, `sp` and `embar` are included in our results.

Eight programs comprise our benchmark set for timing evaluation: `embar` and `sp` are benchmarks for which we expect no improvement—`embar` has no opportunities for optimization, and `sp` comes from the ZPL group's standard benchmark set, and so has already been highly hand-optimized (including our transformation). `pde1` has a few parameters that can be transformed, but they are only in the initialization code, so we expect to see little speedup. `spuv` is `sp` with the hand-optimization undone. `test3` is a tiny program contrived especially to show off our optimization. `frac`, `fibro`, and `zray` are programs outside the ZPL benchmark set that we expect will benefit from the optimization.

Table 1 summarizes the results for compilation time on a DEC Alpha. It is interesting to note that while our pass itself takes little time, the temporaries it inserts can greatly affect the running time of a later pass called *contraction*. This pass attempts to determine which temporaries are unnecessary and remove them. It is this contraction pass which produces the large compiler slowdowns on the `fibro` and `zray` benchmarks. Since the temporaries inserted by our pass are always necessary (the program is not legal without them), compiler performance could perhaps be improved in the future by setting a flag on our temporaries that contraction can check for, enabling it to skip over our temporaries without further analysis.

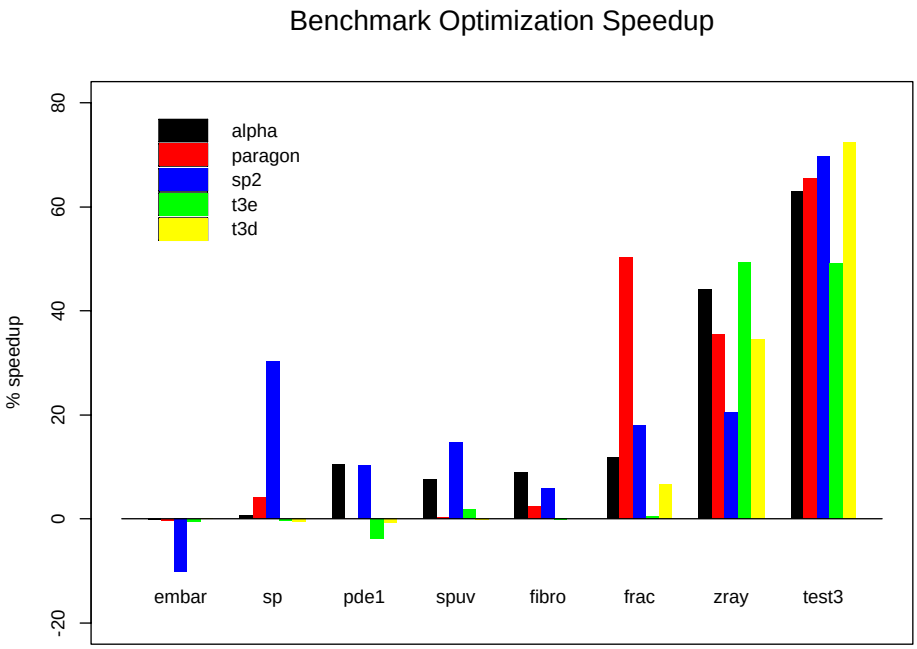
Benchmark Optimization Speedup
% speedup
-20
0
20
40
60
80
alpha
paragon
sp2
t3e
t3d
embar
sp
pde1
spuv
fibro
frac
zray
test3

Figure 4:

Figure 4 shows the optimization speedup attained on the DEC Alpha, a sequential machine, and four parallel machines: the Intel Paragon, the IBM SP/2, and the Cray T3D and T3E. A complete listing of execution results is located in Appendix A. Recall that for `embar` and `sp` speedup is not expected. We include these two programs to validate that our optimization is not grossly harming programs that are not expected to benefit from our transformation.

As expected, we see moderate (5–15%) speedups on `pde1`, `spuv` and `fibro` and larger speedups for `frac`, `zray` and `test3`, the programs that frequently pass large arrays by value.

4.2 Qualitative Evaluation

To provide a qualitative evaluation of the framework, we presented it at a ZPL group meeting and collected feedback. Also a member of the ZPL group is implementing an interprocedural analysis using the framework. The comments we have received outline directions for future work and highlight areas where we have been successful in easing the difficulty of implementing an interprocedural analysis.

In general, the feedback on the framework has been very positive. The group was pleased that the framework, "takes out the hardest part of interprocedural analysis which is handling the recursion." They also observed that `traverse_callsites()` handles the common case of finding callsites in procedures.

Our volunteer implementer re-wrote the basic part of an existing analysis in about an hour, and commented that, although it would probably take over an hour, writing an analysis from scratch would be much easier with our framework. He felt that the main advantage of the framework was that rather than debugging six different analyses, if all of them were written with our framework, potentially only one piece of code would have to be debugged (if the clients of the framework were correct).

His main suggestion for future work was extending the framework to hide statement and expression traversal from the user. Generic routines (`traverse_expression()`, `traverse_stmt()`) are already provided by the compiler but they are clumsy to use because they do not know about contexts.

5 Conclusion

We have implemented an interprocedural framework for the ZPL compiler that facilitates implementing client analyses. We have also added client code that performs alias and parameter in/out analysis. We use parameter in/out analysis to change formal parameters to call by reference when possible.

We believe that parameter in/out analysis is a valuable addition to the compiler. Naive programmers may not understand the advantage of passing parameters by reference and since ZPL defaults to call by value, these programs have the potential to benefit from our transformation.

Our analysis allows users to use the type of parameter passing as documentation, using the `var` tag only when semantically appropriate rather than as a way to improve performance.

The ZPL group is very interested in our framework. The framework not only eases the process of writing an interprocedural analysis, but also leads to more structured and readable client analyses.

We are pleased with the results from our optimization and feel our framework will be a useful tool for implementing future interprocedural analyses in the ZPL compiler.

6 Acknowledgments

We would like to thank all members of the ZPL Development team for their support and enthusiasm about this project. Special thanks go to Brad Chamberlain for his suggestions and debugging help throughout the project and to Sung-Eun Choi for running our test programs on the parallel machines. We would also like to thank E. Christopher Lewis and Jason Secosky for providing ZPL test programs. Finally, thanks to Professor Larry Snyder for his support and encouragement of the project from the very beginning.

References

[CDG96] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical Report UW-CSE-TR 96-11-02, Department of Computer Science and Engineering, University of Washington, 1996.

[Lin96] Calvin Lin. *ZPL Language Reference Manual.* Department of Computer Sciences, University of Texas at Austin, Austin, TX, October 1996. Version 1.0.

[Sny96] Lawrence Snyder. *A ZPL Programming Guide.* Department of Computer Science and Engineering, University of Washington, Seattle, WA, October 1996. Version 4.2.

A Execution Times

Benchmark	Execution Time		
	Unoptimized (s)	**Optimized (s)**	**Speedup**
frac	44.206	38.987	11.8 %
fibro	1.701	1.547	9.1 %
test3	4.081	1.513	62.9 %
pde1	73.946	66.207	10.5%
sp	78.443	77.826	0.8 %
spuv	78.612	72.680	7.5 %
zray	33.323	18.598	44.2 %
embar	6.513	6.523	-0.1%

Table 2: Execution times on a DEC Alpha in CPU seconds with and without our optimization: alias analysis, parameter in/out analysis, and transformation. The cpu is a 21064.

Benchmark	Execution Time		
	Unoptimized (s)	**Optimized (s)**	**Speedup**
frac	13.437	12.546	6.6%
fibro	-	-	-
test3	1.385	0.382	72.4%
pde1	42.948	43.284	-0.8 %
sp	49.904	50.203	-0.6%
spuv	50.099	50.203	-0.2%
zray	44.932	29.457	34.4%
embar	7.823	7.813	0.1%

Table 3: Execution times on the Cray T3D in CPU seconds with and without our optimization: alias analysis, parameter in/out analysis, and transformation. The processors are DEC Alpha 21064s, 150 MHz, dual-issue, with timer granularity 150 ns. It is located at the Arctic Region Supercomputing Center, Fairbanks, AK. Note we were not able to run fibro on the T3D.

Benchmark	Execution Time		
	Unoptimized (s)	Optimized (s)	Speedup
frac	12.498	12.430	0.5 %
fibro	28.589	28.633	-0.2%
test3	0.605	0.307	49.3%
pde1	27.351	28.399	-3.8%
sp	34.203	34.306	-0.3%
spuv	35.005	34.363	1.8%
zray	26.220	13.284	49.3%
embar	4.605	4.626	-0.5%

Table 4: Execution times on the Cray T3E in CPU seconds with and without our optimization: alias analysis, parameter in/out analysis, and transformation. The processors are DEC Alpha 21164s, 300 MHz, quad-issue, with a clock register that is very accurate in ns. It is located at the San Diego Supercomputer Center, San Diego, CA.

Benchmark	Execution Time		
	Unoptimized (s)	Optimized (s)	Speedup
frac	127.881	63.562	50.3 %
fibro	348.755	340.302	2.4 %
test3	5.092	1.757	65.5%
pde1	113.120	113.130	-0.01 %
sp	122.322	117.206	4.2%
spuv	122.769	122.450	0.3%
zray	121.619	78.435	35.5%
embar	13.499	13.537	-0.3%

Table 5: Execution times on the Intel Paragon in CPU seconds with and without our optimization: alias analysis, parameter in/out analysis, and transformation. The processors are Intel i860 XPs, 50 Mhz, quad issue with timer granularity 100ns. It is located at the Dept. of CSE, University of Washington, Seattle, WA.

Benchmark	Execution Time		
	Unoptimized (s)	**Optimized (s)**	**Speedup**
frac	20.371	16.710	18.0%
fibro	32.551	30.618	5.9%
test3	2.040	0.619	69.6%
pde1	30.499	27.380	10.2 %
sp	37.214	25.953	30.3 %
spuv	27.916	23.827	14.6 %
zray	31.476	25.021	20.5%
embar	5.896	6.494	-10.1%

Table 6: Execution times on the IBM SP/2 in CPU seconds with and without our optimization: alias analysis, parameter in/out analysis, and transformation. The processors are RS/6000s, 66.7 MHz, quad issue with a clock register that is very accurate in ns. It is located at the Cornell Theory Center, Ithaca, NY.